10.

MP3 Player

| What You Are Building |
| What Is Progressive Download and What Is Streaming? |
Sound Compression for Streaming	Setting Up	Loading the MP3
What Is the Sound Class?	Stopping and Playing the Music
Displaying the ID3 Information	Changing Tracks
Creating a MP3 Preloader and Displaying the Playback Progress
Changing the Volume

la_eyeworks

Macromedia Flash MX 2004
Beyond the Basics

In this chapter you will learn how to build a nifty, multitrack MP3 player that will play MP3s as it progressively downloads them. You'll build a small interface that will allow the viewer to stop or play the MP3 as well as skip to the next or previous tracks. This MP3 player will also display the artist and track name of the currently playing MP3 using the ID3 tags that are built into the MP3s themselves. Last, you will build a slider that displays the progress of the currently playing MP3, and you'll integrate a preloader into the progress bar as well. As you're probably beginning to realize, you'll be building and learning many new things in this chapter. If you haven't taken a break in a while, now would probably be a good time. ;-)

I. What You Are Building

In this exercise, you will preview and interact with the finished piece you will be building in this chapter. By first having a good understanding of what you will be building, it will help you to better understand some of the more abstract ActionScript topics as you learn them.

1. Open your preferred Web browser, and access the following URL:

http://www.lynda.com/flashbtb/laeyeworks/

2. When the Web site finishes loading, click the **Play** button at the bottom left of the page. This will automatically begin downloading an MP3 music file from the Web server. You'll also notice, when you click the **Play** button, that it turns into a **Stop** button. This is often called a "toggle button" because it toggles back and forth between two different functionalities. It's a great way of conserving space and is something that you see repeated in many other programs.

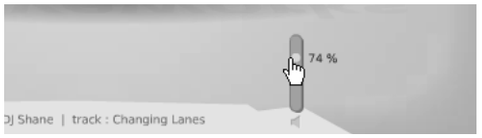
*As soon as a small amount of the MP3 has downloaded to your computer, you'll see two things happen. First, you'll see a light-blue bar animate across the dark-blue bar that sits to the right of the Previous Track/Play/Next Track buttons. This animating, light-blue bar represents how much of the MP3 has been downloaded to your computer. When the light-blue bar reaches the end of the dark-blue bar, the entire MP3 has been downloaded. This is a preloader, much like the preloader you constructed in Chapter 8, "*Building a Preloader*", except this preloader was constructed specifically for the music player. On top of the blue bar is a lighter-blue "dot" that is initially resting on the left side of the bar. But when enough of the MP3 has downloaded to your computer, the music will automatically start playing and the dot will start moving. This dot represents the playback progress of the music, much like the playhead does in Flash MX 2004, Windows Media Player, QuickTime, and so forth. When the dot gets to the right side of the blue bar, the music track has finished playing.*

*Second, after a small amount of the MP3 has been downloaded to your computer, you'll see some text appear to the right of the blue bar. This text states the artist name as well as the track name (in this case, it's me. *grin*) As was mentioned in the introduction to this chapter, the artist and track information is dynamically grabbed from the ID3 tags (more on those later) from the MP3s themselves.*

If you wait until the first track is finished, you'll notice that it will automatically begin downloading another—different—MP3. The preloader will start animating again, and once the music starts playing, the dot will start animating as well. The ID3 track information will also update to display information about the new track. Pretty nifty! The other way to advance to the next track, besides waiting for the current track to play all the way through, is to click the ***Next Track*** *button. Clicking the* ***Previous Track*** *button will, of course, play the previous track in sequence.*

The MP3 player is constructed dynamically in this manner, and streams external MP3 files, because it makes it extremely *easy to update. Whenever you want to add, remove, or change MP3s, you simply make sure they're named appropriately and their ID3 tags are set correctly, drop them in the* ***mp3s*** *folder, change some variables in an external text file, and you're done.*

3. Last, click the **speaker** icon toward the bottom-right side. When you do, you'll notice a small, green slider appear above it.



If you click the light-green dot in the slider, you'll hear the volume of the music change as you drag the slider up and down. Also, to the right of the dot will appear a number that follows the dot as you drag it. That number represents the percentage of the volume level. As you drag the dot up and down, you'll also notice that the text's alpha changes. The louder the volume, the more opaque the volume text. The quieter the volume, the more transparent the text becomes. When you release your mouse button, the percent number will disappear.

So as you can see, there's a lot *of functionality crammed into the bottom portion of the L.A. Eyeworks Web site. In this chapter, you will learn how to build all of that functionality using some actions that you've already learned as well as some that you haven't.*

What Is Progressive Download, and What Is Streaming?

The MP3s that the MP3 player will download off of the Web server will start playing *as they are downloading*. Flash MX 2004 SWF files perform this same way. If you have a longish SWF file, when the viewer comes to your Web site to view your movie, the SWF file will start playing as soon as the first frame has been downloaded. Flash MX 2004 downloads its content–SWFs, MP3s, FLVs (**Fl**ash **V**ideo), JPGs, and so forth–using something called "progressive downloading." As it sounds, a *progressively* downloading file will download and play progressively from the beginning of the file to the end. Once an entire SWF, MP3, or FLV file has been downloaded to the viewer's computer, you are able to "seek" (fast-forward and rewind) through that content with no difficulties. However, if you want to seek to a point in an MP3 file–for example–that hasn't downloaded yet, you're out of luck.

An alternate way of delivering your content to the Flash Player 7 is by using something called "streaming." This is referred to as RTMP Streaming (**R**eal **T**ime **M**essaging **P**rotocol). When content is *streamed* off of the Web server, it is *not* saved to the viewer's computer hard drive. This means that less memory and hard drive space is required to play the content (MP3s, FLVs, and so forth), and because the content is not saved to the viewer's computer, it is considered to be fairly *secure* as well (meaning that it would be difficult for average computer users to take the content that they just watched and use it for their own purposes). With streaming content, such as an MP3 or FLV file for example, you can even "jump" to a portion of the content that hasn't downloaded yet–a feature that a progressively downloading asset doesn't have. However, to truly *stream* content you need special server software called the **F**lash **C**ommunication **S**erver (FCS). You can purchase and read more information about the FCS on Macromedia's Web site:

http://www.macromedia.com/software/flashcom/

The FCS offers a number of advantages over "traditional" *progressive* downloading, such as being able to dynamically serve different FLV files based on the viewer's Internet connection speed. If you're in charge of serving *lots* of video streams to *many* viewers, you should look into the FCS because it offers the most features and flexibility. In Appendix C, "*Getting Your Work Online/Using CGIs*," I will list some Web presence providers that offer the FCS for use.

However, because of the specialized nature and relative complexity of the FCS, this book will cover accessing content using progressive download, which *does not* require any special server software or configuration.

Sound Compression for Streaming

In the MP3 player that you will be building in this chapter, the MP3 sound files will reside–external to the SWF file that will eventually control them–on the remote Web server. You will write some ActionScript that loads and plays one of the MP3 tracks when an event occurs. But how did those music tracks become MP3s? How were the MP3s compressed, and how should one compress MP3s that are going to be progressively downloaded from a Web server?

MP3s can be created using many different programs. In Appendix B, "*Macromedia Flash MX 2004 Resources*," I will address the common programs that you can use to convert a sound file–such as WAV, AIFF, and so forth–into an MP3. For now, however, if you need a good, free, cross-platform program that can play and compress MP3 files, try using Apple Computer's iTunes. It's available for free at **http://www.apple.com/itunes/** and works on both Mac and Windows computers.

As was mentioned in Chapter 2, "*Where Do I Start*?," the target audience is really going to be the defining factor that will determine what level of MP3 compression you should use and even how heavily the video should be compressed, as you'll see in the next chapter. As was discussed in Chapter 2, the target audience for the L.A. Eyeworks Web site is likely to have a broadband Internet connection. Because of that, you can be a little liberal with how the sound files are compressed. In other words, you are able to sacrifice file size–to a point–in exchange for good sound fidelity. However, if most of your target audience still accessed the Internet using a slow dial-up connection, you would instead want to lower the sound quality (which means, to increase the MP3 compression) in exchange for a smaller file size. In essence, the larger the file size of the MP3, the better the audio will sound, but it will take longer to download. Conversely, the smaller the file size of the MP3, the worse the audio will sound, but it will take less time to download. Again, determining the target audience for the Web site (or widget) that you are constructing is *crucial* to deciding what type of MP3 compression to apply to your sounds. Without getting into the nitty-gritty details of MP3 sound compression, bit rates, sample rates, and so forth, here is a short list of MP3 tips that you should be aware of:

- When you have your original music file (WAV, AIFF, and so forth), make sure–in whichever sound-editing program you're using–that the **sample rate** is in an increment of **11 kHz** (11, 22, 44, and so forth). If you use a sample rate that is *not* in an increment of 11, Flash MX 2004 will resample

them when played. This might cause the MP3 to play/behave unexpectedly. Normally, sound files have an 11 kHz incremental sample rate, so this isn't an issue to worry much about. However, if you are integrating audio into your Flash MX 2004 project, and the audio starts sounding like it's in a different pitch, or you notice other strange anomalies, check the audio sample rate to make sure that it's in an increment of 11 kHz.

- When choosing an MP3 **bit rate** to encode your sound file to, keep in mind that the bit rate corresponds to the minimum bandwidth throughput that the viewer will need to download the MP3 at a fast enough rate so that the playing of the sound doesn't overtake the downloading of the sound. In other words, if you encode your sound file with a **bit rate** of 126 kbit/sec, the viewer downloading that MP3 will need to have an Internet connection of *at least* 12.6 k/sec or greater to download the sound as fast or faster than it is playing. As long as the viewer has a connection speed that is equal to or greater than the bit rate of the MP3, the sound will not stop or "stutter" as it is downloading. The MP3s you will be using for the L.A. Eyeworks Web site are encoded using a **v**ariable **b**it **r**ate (VBR), which means that the bit rate will increase or decrease to accommodate how dynamic the sound is. Averaged out, the bit rates are approximately 103 kbit/sec, meaning that the MP3s will stream uninterrupted over a Internet connection of at least 10.3 k/sec. (The average dial-up modem speed is 5.6 k/sec. Broadband is usually *at least* 10 times that amount.) However, Flash MX 2004 has an action that will allow you to specify how much of an MP3 it should buffer (download and cache) before it begins playing the sound file. This will help to decrease the possibility of the sound playing faster than it can download, which results in the sound stopping before it reaches the end. You will learn about that action later in this chapter.
- When assigning or modifying the ID3 tags on your MP3s, use only ID3 v2.3 or v2.4 tags. v2.2 tags *are not supported* and ID3 v1 tags are inserted at the *end* of the MP3 file (which means they cannot be accessed or used until the MP3 has been *completely* downloaded). ID3 v2.3 or v2.4 tags are stored at the *beginning* of the MP3 file and can therefore be accessed as soon as the MP3 starts downloading. You'll learn more about ID3 tags later in this book, and a list of programs that can view and edit MP3 ID3 tags are listed in Appendix B, "*Macromedia Flash MX 2004 Resources*."

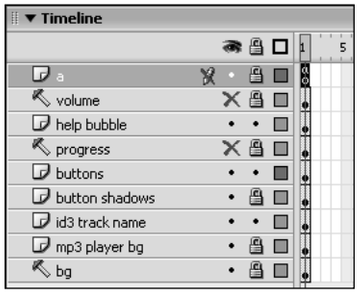
Note: If you'd like to read more about how Fraunhofer MP3 (the MP3 compression scheme that Flash MX 2004 supports) compression works, you can read more about it here:

http://www.iis.fraunhofer.de/amm/techinf/layer3/index.html

2. Setting Up

In this exercise, you will make a few modifications to the prebuilt starter **music.fla** file that you will be working with for the remainder of this chapter. Just like the previous chapters, this chapter includes an FLA that has some prebuilt elements and ActionScript to allow you to jump right into the new exercises and steps, bypassing the material you've already learned. In this exercise, you will also get a brief tour of what has been included with the provided FLA so that you have a good understanding of what you will be building on.

1. Navigate to your **Desktop > la_eyeworks > site** folder and double-click **music.fla** to open it in Flash MX 2004.



2. Once the FLA opens, look at the **Timeline**. You might need to expand the Timeline view a little to see all the layers.

*As you can see, there's quite a lot of provided content in this FLA. You'll also notice—besides the usual **bg** layer—that two layers (**volume** and **progress**) are specified as guide layers, and are locked and hidden. These layers are set like this so they will not be visible while you're working and will not be exported when you test your movie. This is because these layers are there for you to use when watching the included movies **volume_slider.mov** and **music_progress.mov**, which you'll read about later in this chapter. Until then, however, these layers are essentially disabled so they'll stay out of your way while you work on the other exercises.*

3. Click **keyframe 1** in layer **a**, and open the **Actions** panel (**F9**).

```
/*
//----------------<pre-built ActionScript for Chapter 10>-------------------\\

//----------------<sound initialization>-------------------\\
var curTrackNum:Number = 0;

// autosize some text fields
this.helpBubble.autoSize = "center";
this.trackInfo.autoSize = "left";

// load the track info vars
var myMusicLv:LoadVars = new LoadVars();
myMusicLv.load("vars/track_info.txt");

//----------------</sound initialization>-------------------\\

//----------------<volume control>-------------------\\
```

Note: *Not all of the ActionScript provided with* ***music.fla*** *is pictured in the code above. Because of the large amount of included ActionScript, it would be impractical to picture it all here.*

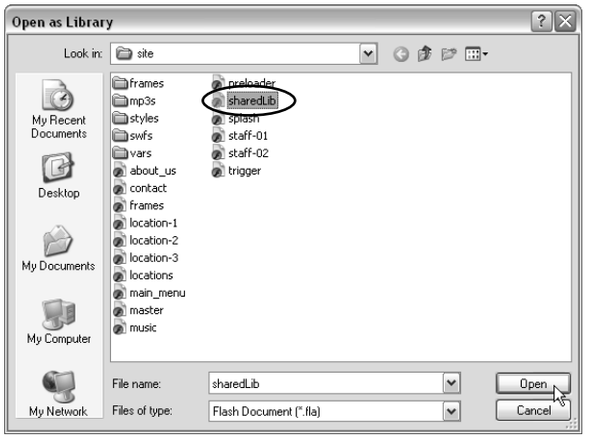
Just like a few of the previous prebuilt FLAs, you'll notice that this FLA comes with quite a lot of ActionScript, already written into the first keyframe. The included actions are a testament to how much you've learned so far in this book, because the actions you see in that first keyframe are all actions you already know. If you scroll down the Actions panel, you should see lots of familiar actions there. Although you probably do not immediately understand what they're all doing, you'll be pulling some of those actions out of the commented area to use in various exercises throughout the remainder of this chapter.

4. Minimize the **Actions** panel by pressing **F9** again.

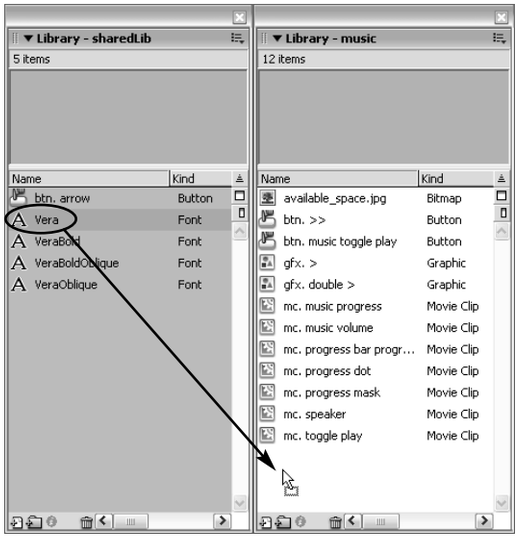
At the bottom of the Stage, you'll notice two dynamic text fields. The small one above the Play/Stop, Next Track, and Previous Track buttons is where a small amount of descriptive text will appear when the viewer rolls his or her mouse over the buttons beneath it. This dynamic text field already has an instance name of ***helpBubble*** *applied to it. The larger dynamic text field toward the bottom-middle of the Stage is where the MP3 ID3 information will be displayed. This field already has an instance name of* ***trackInfo*** *applied to it. As with many of the other prebuilt FLAs you've worked with in this book, in the next few steps you will import a shared font and apply it to those text fields.*

5. First, make sure you *do not* have **sharedLib.fla** open. If it is open, the next few steps will not work correctly.

6. Choose **File > Import > Open External Library**.



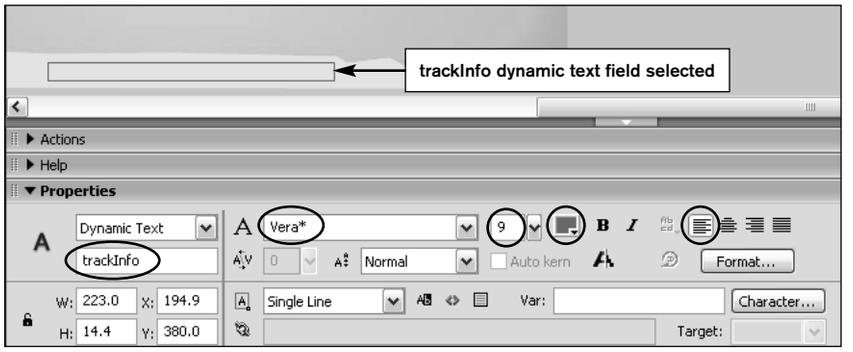
7. In the **Open as Library** window, navigate to your **Desktop > la_eyeworks > site** folder and single-click **sharedLib.fla** to select it. Click **Open** to open only the **Library** of **sharedLib.fla**.



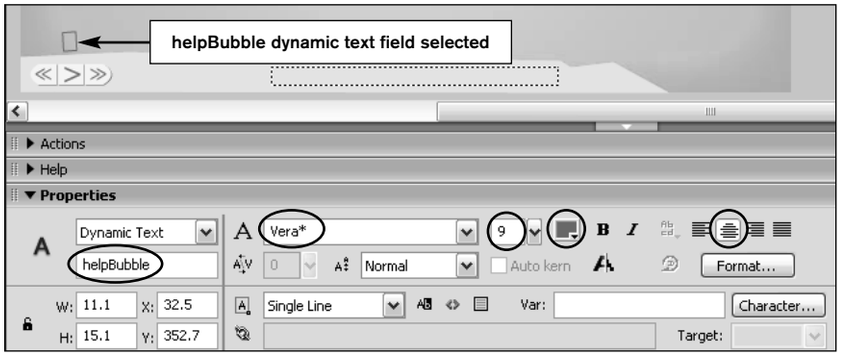
8. Open the **music.fla Library** window (**Ctrl+L** [Windows] or **Cmd+L** [Mac]) and position the **sharedLib Library** and the **music Library** windows next to each other so they can both be seen simultaneously. Then, drag the shared font **Vera** from the **sharedLib Library** window onto the **music Library** window.

*As you have done a few times before in this book, you just made a link to the shared font **Bitstream Vera Sans** in **music.fla**. You don't need to drag over any other shared fonts because Vera is the only font that will be used in the construction of the MP3 player.*

9. Close the **sharedLib Library** window and redock the **music Library** window if need be.



10. Next, single-click the large dynamic text field on the Stage, **`trackInfo`**. Then, in the **Properties inspector**, click the **Font** pull-down menu and choose the shared font **Vera***. The remaining options have already been set for you, but quickly double-check that what you see in your **Properties inspector** is the same as the selected options in the screen shot above.



11. Repeat Step 10, but this time make sure you have the small, bottom-left dynamic text field (**`helpBubble`**) selected first. Once again, verify that the settings you see in the **Properties inspector** match those in the screen shot above.

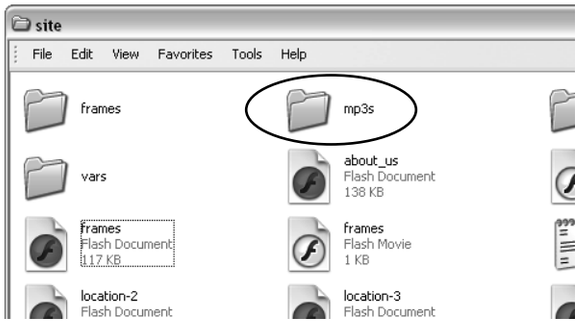
That's it! The symbols that you see on the Stage—and even those that are on the locked and hidden guide layers—already have instance names applied to them to save you time and energy. Don't worry, you'll be using that saved energy later in this chapter as you write the ActionScript to construct this über cool MP3 player.

In the next exercise, you will write the ActionScript that dynamically loads and plays the first MP3.

3. Loading the MP3

In this exercise, you will write the ActionScript that will load and play the first external MP3. As you work through the steps in this exercise, you'll see many similarities between how this ActionScript is written and constructed and how some previous actions have been authored. In the following exercises in this chapter, you will learn how to give the viewer the ability to change the currently playing MP3 track, stop or play the music, observe the download and playing progress of an MP3, and change the volume of the music. This is simply the first–but critical–step in building the entire, fully functional MP3 player.

1. Hide or minimize Flash MX 2004. On your hard drive, navigate to your **Desktop > la_eyeworks > site** folder.



2. In the **site** folder you'll see–among other things–a folder called **mp3s**. Double-click that folder to open it.

*In the **mp3s** folder, you'll see two MP3s, **mp3-0.mp3** and **mp3-1.mp3**. These are the two MP3 tracks that you will be loading and controlling in this chapter. Now that you know what they're named and where they're located in relation to the rest of your site files, you can write the ActionScript to load and manipulate them using the **Sound** object.*

3. Close the **mp3s** folder and return to Flash MX 2004. Make sure **music.fla** is still the foreground FLA.

4. Select **keyframe 1** in the **a** layer and open the **Actions** panel (**F9**).

*The first step is to create a **Sound** object to handle the music that you will load and later control. The sound object, as you are about to see, is very similar in construction to the **MovieClipLoader** and **LoadVars** scripts you've built in previous chapters.*

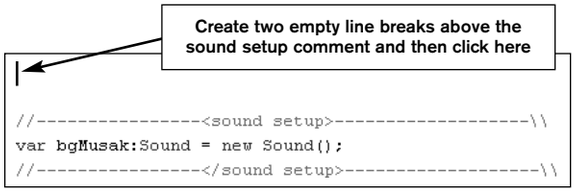
```
//----------------<sound setup>-------------------\\
var bgMusak:Sound = new Sound();
//----------------</sound setup>------------------\\

/*
//----------------<pre-built ActionScript for Chapter 10>-------------------\\

//----------------<sound initialization>-------------------\\
var curTrackNum:Number = 0;
```

5. Click the empty line break–already provided for you–at the top of the **Actions** panel above the prebuilt actions. Then, create a new sound object by typing the following:

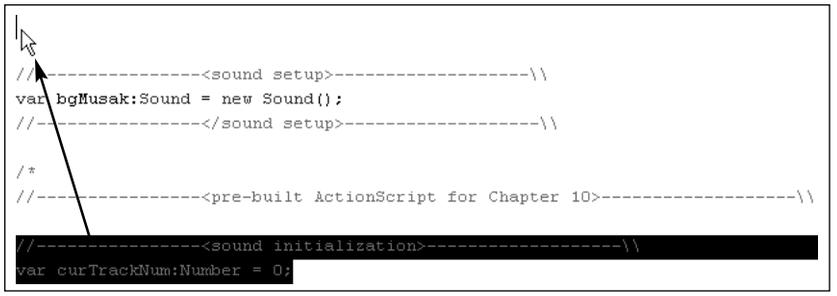
```
//----------------<sound setup>----------------\\
var bgMusak:Sound = new Sound();
//----------------</sound setup>----------------\\
```

*As you have already seen before with the **MovieClipLoader** and **LoadVars** objects, the action you just wrote simply creates a new **Sound** object named **bgMusak** under a commented line.*

*Much like the slideshow you constructed in Chapter 7, "*Building a Slideshow*," the viewer will be able to advance forward and backward through the available MP3 tracks. However, where the slideshow had 10 images, the MP3 player in this chapter will use only two tracks. As you'll see as you write the ActionScript for the MP3 player, it's very easy to add or remove MP3 tracks at a later time. Because the MP3 player will allow the viewer to switch between tracks, you need to create a variable to keep track of which MP3 track is currently playing, just like you did with the **curFrameNum** variable you created for the slideshow.*



6. Click at the beginning of the `//----------------<sound setup>----------------\\` comment line and press **Enter** (Windows) or **Return** (Mac) twice to push all the actions in **Keyframe 1** down two lines. Then, click at the new, empty line at the top of the **Actions** panel.

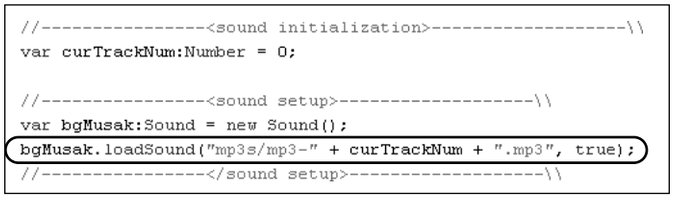


7. Select the following two lines of actions:

```
//----------------<sound initialization>----------------\\
var curTrackNum:Number = 0;
```

Drag them to the topmost empty line break you created in Step 6. This moves the action and comment line out of the commented-out prebuilt actions that were included with the FLA, thereby making them "active."

Although the comment line is self-explanatory, you probably want to know what the ***var curTrackNum:Number = 0;*** *action is for. Again, just like you used the variable* ***curFrameNum*** *in the slideshow FLA to keep track of the current slide number, this* ***curTrackNum*** *variable will keep track of the currently playing MP3. Because you will script the capability of the MP3 player to change tracks, you need to be able to keep track of* which *track is currently playing. Later, you will use the number within that variable (****0****) to specify which MP3 track should be loaded.*



8. Click at the end of the **`var bgMusak:Sound = new Sound();`** line, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and type the following:

```
bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);
```

*Although you may initially be somewhat confused as to what this action is doing, keep in mind that this action is very similar to the action you set up in Chapter 7, "*Building a Slideshow*," Exercise 3, Step 6, that allowed you to load a JPG file into the slideshow. This action you just wrote loads an MP3 using the* ***Sound*** *object* ***loadSound*** *event handler. The name of the MP3 that it loads starts with the string "mp3s/mp3-" (because the MP3 name starts with "mp3-" and is in a folder titled* ***mp3s*** *in your* ***Desktop > la_eyeworks > site*** *folder). After that string, the script takes the number in the* ***curTrackNum*** *variable,* ***0****, sticks it onto the "mp3s/mp3-" string that came before it and the ".mp3" string that comes after it. The final result of that action is* ***mp3s/mp3-0.mp3****. This is, of course, the path to—and the name of—the first MP3 that will load and play.*

You'll also notice that this action has ***true*** *written into it. This Boolean value specifies whether the loading MP3 should be considered a* ***streaming*** *sound (****true****), or not (****false****). A streaming sound, as this MP3 player uses, will start playing as soon as enough of the sound has been downloaded and will continue to play as the remainder of it is downloaded. A nonstreaming sound is considered to be an* ***Event*** *sound and will* not *begin playing until the* entire *sound has been downloaded to the viewer's computer.*

Since you have now written enough ActionScript to create a wee bit of functionality, you can test your movie!

---

9. Save the changes you've made to **music.fla** by choosing **File > Save** (**Ctrl+S** [Window] or **Cmd+S** [Mac]).

---

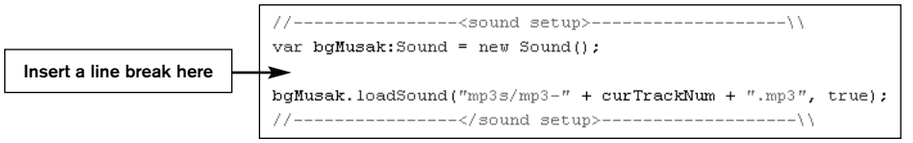
10. Test your movie by choosing **Control > Test Movie** (**Ctrl+Enter** [Windows] or **Cmd+Return** [Mac]).

*When the Preview window opens, you won't see a whole lot, but you'll hear the first MP3 (****mp3-0.mp3****) immediately start playing. When the track finishes playing, it will stop. In Exercise 5 in this chapter, you will write a script that will automatically instruct the MP3 player to load and play the next MP3 in sequence when the current MP3 finishes playing. Congratulations! You've just loaded and streamed an external MP3.*

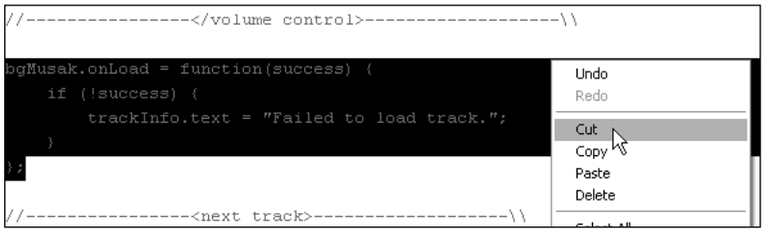
Last, you should write a small script that shows the viewer an error message if an error is encountered when loading an MP3.

---

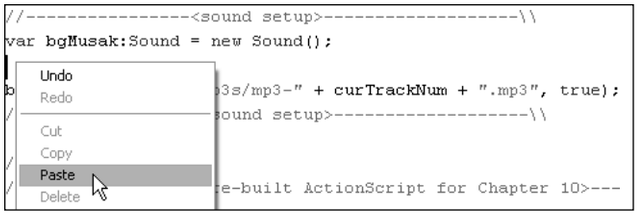
11. Close the **music.swf Preview** window and return to **music.fla**. Make sure you have **Keyframe 1** selected in layer **a**, and make sure the **Actions** panel is open.



12. Click at the end of the action **`var bgMusak:Sound = new Sound();`** and press **Enter** (Windows) or **Return** (Mac) to create a line break. This is where you will write a script that will insert an error message into the **`trackInfo`** dynamic text field if there's an error when attempting to load an MP3.



13. Scroll a little over halfway down the **Actions** panel until you get to the script you see selected in the screen shot above. The script resides between the **`//----------------</volume control>----------\\`** comment and then **`//----------------<next track>----------------\\`** comment. Select the script and cut it out of the **Actions** panel by **right-clicking** (Windows) or **Ctrl+clicking** (Mac) anywhere on that selected script, and choosing **Cut** from the contextual menu.

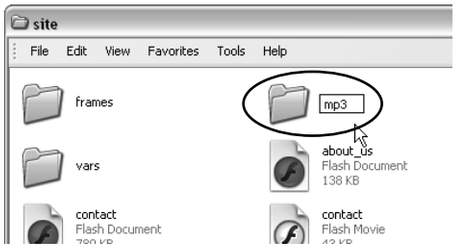


14. Scroll back to the top of the **Actions** panel, and **right-click** (Windows) or **Ctrl+click** (Mac) in the empty line break you created in Step 12. From the contextual menu that appears, choose **Paste**. This will paste the script–which you cut in Step 13–so that it comes immediately after the **`var bgMusak:Sound = new Sound();`** action.

It's important that this pasted script comes before the ***`bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);`*** *action. In this case, the error message won't display unless the* ***`bgMusak.onLoad`*** *action comes* before *the* ***`bgMusak.loadSound`*** *action.*

*The **bgMusak.onLoad** script you just cut and pasted was included for you in the provided **music.fla** because it is nearly identical to–and has the same functionality–as the **myLV.onLoad** script that you wrote in Chapter 4, "*LoadVars Class*," Exercise 3, Steps 8 – 14. Where the **myLV.onLoad** script showed an error message to the viewer if the variables could not be loaded from an external text file, the **bgMusak.onLoad** script shows the viewer an error message if an external MP3 cannot be loaded. The **bgMusak.onLoad** script essentially reads, "When the MP3 loads, if it was not successful (**!success**) in its loading, insert the text 'Failed to load track.' in the **trackInfo** dynamic text field on the Stage."*

Of course, you would be a slack Flash developer if you didn't test that error handling to make sure it works correctly.

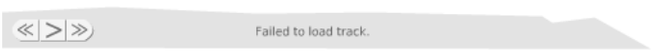


15. Hide or minimize Flash MX 2004. On your hard drive, navigate to your **Desktop > la_eyeworks > site** folder. Then, rename the **mp3s** folder to **mp3**. (You're just temporarily removing the **s** at the end in order to test your script.)

16. Then, return to Flash MX 2004 and make sure **music.fla** is still your foreground FLA.

17. Save the changes you've made to **music.fla** by choosing **File > Save** (**Ctrl+S** [Window] or **Cmd+S** [Mac]).

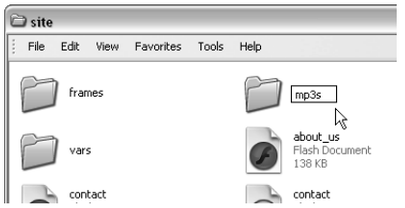
18. Test your movie by choosing **Control > Test Movie** (**Ctrl+Enter** [Windows] or **Cmd+Return** [Mac]).



*When **music.swf** opens in the Preview window, not only will you see the output window open with an error message (close the output window when it opens), but you'll also see the error message "Failed to load track" appear in the **trackInfo** text field that sits on the MP3 player bar at the bottom of the Stage. Who would've thought you'd be happy to see an error message, eh?*

*Now that you know the MP3 loads and plays correctly, and that the error message appears if there's a problem loading the MP3, you can rename the **mp3** folder back to its* correct *name of **mp3s**.*

19. Close the **Preview** window and hide or minimize Flash MX 2004.



20. On your hard drive, navigate to your **Desktop > la_eyeworks > site** folder and rename the folder **mp3** to its correct name, **mp3s**.

---

21. Return to Flash MX 2004 and make sure that **music.fla** is the foreground FLA.

Give yourself a big pat on the back (and maybe a stiff drink) because you just completed the loading, streaming, playing, and error handling of the first MP3.

In the next exercise, you will learn how to stop and play the music. Take a quick break and I'll meet you back here in five minutes!

What Is the Sound Class?

The **LoadVars** class loads and sends variables, the **MovieClipLoader** class loads SWFs and JPGs, and now you have the **Sound** class, which loads sounds. The **Sound** class can either load sounds that are external to the SWF by using the **Sound.loadSound()** method–which you just learned in the last exercise–or it can even control a sound file (with linkage turned on) in a FLA's Library using the **Sound.attachSound()** method. By using the **Sound** class, you can do the following:

- Dynamically load sounds that are external to the SWF itself.
- Monitor the downloading progress of an external sound file.
- Change the volume and pan of a sound.
- Start and stop a sound.
- Read the ID3 tags of an MP3.

The **Sound** class can work with AIFF, WAV, and MP3 file formats. For the most part, however, you'll probably be working mostly with MP3 sound files because they offer a smaller file size with comparable quality. **Note:** The smaller the file size of the media you're working with, the less RAM is consumed on both the author and viewer's computer.

For your reference, here is a table listing the **Sound** class methods, properties, and event handlers. You can read more about the **Sound** class by opening the **Help** panel (**F1**), and in the left section of the **Help** panel choosing **ActionScript Dictionary > S > Sound class**. As you work through the exercises in this chapter, you'll get to use some of these in the construction of the L.A. Eyeworks MP3 player:

Methods, Properties, and Event Handlers of the Sound Class

METHODS

Method	Description
attachSound()	Allows you to specify the linkage name of a sound in the Library that you want to attach to a **Sound** object.
getBytesLoaded()	When this action is executed, it returns the number of bytes that have been downloaded up to that point from the currently loading sound.
getBytesTotal()	When this action is executed, it returns the total number of bytes of the sound that is currently being downloaded.
getPan()	When the sound pan is set using **setPan()**, the **getPan()** method will return what the last value set using **setPan()**.
getTransform()	Similar to **getPan()**, except **getTransform()** returns the last value set using **setTransform()**.
getVolume()	Similar to **getPan()** and **getTransform()**, **getVolume()** returns the last value set using **setVolume()**.
loadSound()	Allows you to load an external MP3. The **loadSound()** method allows you to specify the path to the MP3 you want to load as well as whether it should play while downloading (streaming sound) or not (event sound).
setPan()	Allows you to specify the pan (left and right balance) of a sound attached or loaded into a **Sound** object.
setTransform()	Somewhat of a combination between **setPan()** and **setVolume()**, the **setTransform()** method not only allows you to simultaneously manipulate the panning *and* volume of a playing sound, but also the level of the sound in each ear. This allows you to achieve some fairly complex panning and volume effects with your sounds.
setVolume()	Allows you to specify the volume of a sound attached or loaded into a **Sound** object.
start()	Allows you to start playing a sound. The **start()** method also allows you to specify a few parameters such as how many seconds into the sound it should start playing, as well how many times you want the sound to loop.
stop()	If you just use the **stop()** method with no parameters, all sounds will stop playing. But you can also specify the linkage ID name like so: **stop("myLinkedSymbol")**, which stops the playback of a single sound.

continues on next page

Methods, Properties, and Event Handlers of the Sound Class *continued*

PROPERTIES

Property	Description
duration	Returns the duration of a sound in milliseconds. **Warning:** If you use the **duration** property to find out the duration of a linked sound, it will return the actual duration of the sound. However, if you use the **duration** property to find out the duration of a sound that is currently being streamed, it will return only the duration of the sound that has been downloaded *up to that point*.
id3	Allows you to access the ID3 information of an MP3 file. You will learn more about this property in Exercise 4.
position	Returns the number of milliseconds that a sound has been playing. You can use this property in conjunction with the **duration** property to create a playback progress bar.

EVENT HANDLERS

Event Handler	Description
onID3	Each time new ID3 information is available, this event handler (and any actions you place within it) is executed. This is useful for triggering the population of a text field with MP3 ID3 information.
onLoad	Executed when a sound loads. You used this in the last exercise to display an error message if the MP3 is unable to load correctly.
onSoundComplete	Executed when the MP3 has stopped playing. You will use this event handler in Exercise 5 to tell the Flash Player 7 to automatically load the next MP3 track in sequence when the currently playing MP3 reaches the end.

Similar to the **LoadVars** and **MovieClipLoader** classes, the **Sound** class offers a variety of ways to load, monitor, and control sounds. You'll be using a few of these methods, properties, and event handlers throughout this chapter.

4. Stopping and Playing the Music

There's nary a more irritating set of circumstances than when you visit a Flash MX 2004–based Web site, the music starts playing, and you can't turn it off. This shameful act doesn't seem to occur as much as it used to two to four years ago, but every now and then I stumble across a Web site where the forgetful (or just straight-up reckless!) designer declined to give the viewer a way to turn off the looping bit of music. In this exercise, you will show pity on the poor Internet surfer and build a way for him or her to turn your music off should it become too distracting. In case viewers change their minds once the music has been turned off, you're also going to provide them with a way to turn it back on again. Tallyho!

At this point in the construction of your MP3 player, the MP3 starts playing automatically. However, in a new area of politeness, more and more Flash MX 2004 designers and developers are opting to have the music *off* by default. That way, viewers are not aurally bombarded when they first enter your site. Then, if viewers feel that some music would go well with what they're looking at, they can click the Play button to begin playing the music. With your current MP3, the music starts automatically playing because the **bgMusak** sound object actions that control the playing of the music are all sitting out in "the open" in Keyframe 1. An easy way to prevent those actions from being automatically executed when **music.swf** first loads is to enclose them in a function. That way, the actions within the function will be executed only when the function is explicitly called.

So the first step in this process is to create a new function and move the **bgMusak** sound object actions into their new function home.

```
//----------------<sound setup>-------------------\\
function playMusak() {
var bgMusak:Sound = new Sound();
bgMusak.onLoad = function(success) {
    if (!success) {
        trackInfo.text = "Failed to load track.";
    }
};
bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);
//----------------</sound setup>------------------\\
```

1. Click at the end of the comment **//----------<sound setup>----------** line, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and type the following:

```
function playMusak() {
```

*As you have seen before, this is the opening action to create a new function called **playMusak**. Next, you need to specify where the function ends.*

```
//----------------<sound setup>-------------------\\
function playMusak() {
var bgMusak:Sound = new Sound();
bgMusak.onLoad = function(success) {
    if (!success) {
        trackInfo.text = "Failed to load track.";
    }
};
bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);
};
//----------------</sound setup>------------------\\
```

2. Click at the end of the **`bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);`** line, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and type the following:

```
};
```

*This closed curly brace marks the end of the **`playMusak`** function. All the actions between the **`function playMusak() {`** line and the closed curly brace you just typed are now part of the **`playMusak`** function. Now, a few of you might be wondering what the action **`var bgMusak:Sound = new Sound();`** is doing contained within the function. After all, that action doesn't start the music playing. It just simply creates a new **`Sound`** object called **`bgMusak`**. The reason why the action that creates the **`bgMusak`** sound object is contained within the **`playMusak`** function is because every time the viewer clicks the **Next Track** or **Previous Track** buttons, or when the track ends and advances to the next track in sequence, you want the **`bgMusak`** sound object to be re-created again from scratch. By leaving the action that creates the **`bgMusak`** sound object in the **`playMusak`** function, each time the function is executed the **`Sound`** object will be created anew. If you* did not *do this, whenever the MP3 player switched to another track, some of the **`bgMusak`** sound object's properties—such as **`duration`** and **`position`**—would get confused. In their little world, they're buzzing happily along, tracking the duration of the MP3 and the current millisecond position that its played so far, when the duration of the MP3 suddenly changes! If you were using the **`duration`** and **`position`** properties to script a playback progress slider (as you will be building later in this chapter), since those values suddenly changed with the switching of the MP3, the playback slider is all messed up now, too. Drat!*

*By re-creating this **`Sound`** object each time another track is loaded, it will prevent that aforementioned confusion and all the properties associated with that **`Sound`** object will be cleared as well.*

*In an attempt to follow correct ActionScript formatting, the actions within the **`playMusak`** function should be indented.*

```
//----------------<sound setup>-------------------\\
function playMusak() {
    var bgMusak:Sound = new Sound();
    bgMusak.onLoad = function(success) {
        if (!success) {
            trackInfo.text = "Failed to load track.";
        }
    };
    bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);
};
//----------------</sound setup>-------------------\\
```

3. Select all the actions between the **`function playMusak() {`** line and the curly brace that ends that function (**}**). Once all those actions are selected, press **Tab** to indent those actions.

However, there's a slight problem with the way that script is currently set up. In Step 5 of the previous exercise, you created a new ***Sound*** *object by typing* ***`var bgMusak:Sound = new Sound();`****. Then, in Steps 1 and 2 in this exercise, you moved that action, and a few others, into a function. What this means is that the* ***bgMusak*** *variable that you're attaching a new* ***Sound*** *object to resides within the function* ***playMusak*** *and can therefore* only be accessed *from within that same function. Why is this a bad thing? Because if you are writing a script, and that script originates from* outside *of the* ***playMusak*** *function, you can't get access to the* ***bgMusak*** *sound object.*

To get around this, what you need to do is first create the ***bgMusak*** *variable from* outside *the* ***playMusak*** *function, and then from within the function you can assign that variable to the* ***Sound*** *object. That way, when you need to refer to the* ***bgMusak*** *variable that the* ***Sound*** *object is attached to (to tell the sound to stop, get its duration, and so forth) from outside of the* ***playMusak*** *function, you can easily do so. This is called* initializing a variable*.*

```
//----------------<sound setup>-------------------\\
function playMusak() {
    bgMusak = new Sound();
    bgMusak.onLoad = function(success) {
        if (!success) {
            trackInfo.text = "Failed to load track.";
        }
    };
    bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);
};
//----------------</sound setup>-------------------\\
```

4. Within the **`playMusak`** function, change the action **`var bgMusak:Sound = new Sound();`** so that it instead reads as follows:

```
bgMusak = new Sound();
```

However, because you still want to set the data type of the ***bgMusak*** *variable to a* ***Sound*** *object using strict typing, in the next step you're going to initialize that variable* outside *of the function. As covered previously, if you used* ***`var bgMusak:Sound`*** *to strict type the* ***bgMusak*** *variable from* within *the function,* ***bgMusak*** *would then reside* within *the function and would be inaccessible from* outside *of it.*

```
//----------------<sound initialization>-------------------\\
var curTrackNum:Number = 0;
var bgMusak:Sound;

//----------------<sound setup>-------------------\\
function playMusak() {
    bgMusak = new Sound();
    bgMusak.onLoad = function(success) {
        if (!success) {
            trackInfo.text = "Failed to load track.";
        }
    };
    bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);
};
//----------------</sound setup>-------------------\\
```

5. Click at the end of the **`var curTrackNum:Number = 0;`** line toward the top of the **Actions** panel, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and type the following:

```
var bgMusak:Sound;
```

*This action creates a new variable called **`bgMusak`** and strict types it to a **`Sound`** object. So now, the **`bgMusak`** variable resides on **`_root`**,* outside *of the **`playMusak`** function. The action within the function, **`bgMusak = new Sound();`**, now just assigns a new **`Sound`** object to the **`bgMusak`** variable that is on **`_root`**. What this means is that now, when you want to refer to the **`bgMusak`** sound object, you can easily do so by writing **`_root.bgMusak`**, or—if the action you're authoring also resides on **`_root`**—by typing **`bgMusak`**.*

Now that the first MP3 won't automatically start downloading and playing—because you inserted that ActionScript into a function—you can start writing the actions that will play and download the music when the viewer clicks the Play button, and stop the music when he or she clicks the Stop button.

6. On the **Stage**, double-click the instance of the **mc. toggle play** movie clip symbol. This movie clip symbol contains both the **Play** and **Stop** buttons and acts as a toggle button. (When you click the **Play** button, it advances to the next frame to show the **Stop** button. When you click the **Stop** button, it goes back to the **Play** button.)

7. Once you've opened the **mc. toggle play** movie clip symbol, single-click the **Play** button on the **Stage** and open the **Actions** panel (**F9**).

At this point, some of you might be wondering—after all that talk of centralizing *ActionScript code by putting it on the first keyframe—why you're selecting a button symbol and opening the* ***Actions*** *panel? Sacrilegious! The reason is, in this case, to retain simplicity. To save valued space on the Stage, the Play and Stop buttons were made into a toggle button. Just like the controllers for QuickTime and Windows Media Player, when you click the Play button, and the media begins to play, the Play button toggles to a Stop button. Clicking the Stop button stops the media and then toggles back to a Play button.*

There are a few different ways to create toggle buttons; putting some buttons in a movie clip symbol is only one of those ways. When using the technique of adding mouse events to button and movie clip symbols from a keyframe—what you've been using exclusively up until this point—it unfortunately breaks down when it comes to a movie clip–based toggle button. ActionScript can't instruct a symbol to do something if that symbol does not exist yet. In the case of this toggle button, the Play button is initially visible, but the Stop button is not. Simply, this messes up the way you've been writing your ActionScript up until now. Sure, there are ways of making it work, using different techniques, while still keeping your ActionScript centralized, but in this case it's easier to go low-tech than it is to go high-tech. Rather than try to stuff more new techniques and ActionScript down your throat just to create a simple toggle button, I felt that it would be best to go simple in this case. You've already got enough new material to learn in this chapter. :-)

```
on (rollOver) {
    _root.helpBubble.text = "play music";
}

on (rollOut) {
    _root.helpBubble.text = "";
}

on (release) {
    nextFrame();
}
```

*Upon opening the **Actions** panel, you'll notice that this Play button already has some ActionScript assigned to it. All that these actions are really saying is, "When the viewer rolls his or her mouse over the Play button, put the text 'play music' in the **helpBubble** dynamic text field on the Stage. When the viewer rolls his or her mouse off of the Play button, clear the text in the **helpBubble** field. Last, when the viewer clicks the Play button, move the playhead in this movie clip symbol to the next keyframe, thereby displaying the Stop button." The Stop button also has ActionScript already assigned to it that essentially does the opposite of what the Play button does.*

Now, what you need to add is the action that says, "When the viewer clicks the Play button, play the music."

```
on (rollOver) {
    _root.helpBubble.text = "play music";
}

on (rollOut) {
    _root.helpBubble.text = "";
}

on (release) {
    _root.playMusak();
    nextFrame();
}
```

8. Click at the end of the **`on (release) {`** line, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and type the following:

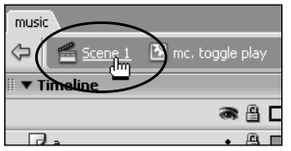
```
_root.playMusak();
```

*This action tells the **playMusak** function—that you moved all the sound-related actions into back in Steps 1 and 2—on **_root** to execute. When that function is executed, the first MP3 track will start downloading and playing.*

Once the music has started playing, you of course want to give the viewer a way to turn it off. However, just telling the music to stop playing won't actually do the trick. The problem with simply telling the music to stop is that if one of the MP3s is currently downloading when the viewer clicks the Stop button, even though you could write an action that told the music to stop—and it would—the MP3 would still continue to download. This means that until that track finishes its downloading, most of the visitor's available bandwidth would be consumed. This will slow down everything else the visitor tries to access/download on your Web site until that downloading file completes. So instead of simply telling the music to stop, you're going to write a quick action that deletes the entire ***bgMusak*** *sound object. By deleting the* ***bgMusak*** *sound object, not only will the sound stop playing, but if an MP3 track is currently in the process of downloading, the download will stop as well!*

At this point, some of you might be saying, "Delete the ***Sound*** *object? Uhh…don't we need that to play the MP3s?" As a matter of fact, you do need the* ***bgMusak*** *sound object to play the music. But remember, with the way the MP3 player is currently written, the* ***bgMusak*** *sound object is created anew every time the Play button is clicked and every time—as you will see later—the Next Track and Previous Track buttons are clicked.*

In the next few steps, you will create a function on the Scene 1 Timeline (_root) that deletes the ***bgMusak*** *sound object. You're putting this sole action within a function because it will be used by the Stop button as well as the Stop button as well as the Next Track and Previous Track buttons.*



9. Return to **Scene 1** by clicking its tab above the top left of the **Timeline**.

10. Click **Keyframe 1** in layer **a** and open the **Actions** panel (**F9**).

```
//----------------<sound setup>-------------------\\
function stopMusak() {
function playMusak() {
    bgMusak = new Sound();
```

11. Toward the top of the **Actions** panel, click after the comment **`//----------------<sound setup>----------------\\`** line, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and type the first line of the function:

```
function stopMusak() {
```

```
//----------------<sound setup>------------------\\
function stopMusak() {
    delete bgMusak;
};
function playMusak() {
```

12. Click at the end of the action you typed in the previous step, press **Enter** (Windows) or **Return** (Mac) once to create another line break, and type the following:

```
delete bgMusak;
};
```

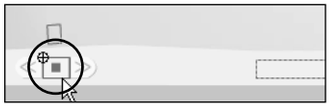
And again, even though you have to type in the line breaks, Flash MX 2004 will automatically handle the formatting (indentation) of the ActionScript for you.

Now you have a function called ***stopMusak*** *that, when called, will delete the* ***bgMusak*** *sound object. Not only will this cause the playing music to stop, but it will also halt any downloads that the* ***bgMusak*** *sound object is performing.*

Warning: *Because of a bug in the current version (current as of this writing is 1.2) of the Web browser* ***Safari*** *that runs in Mac OS X, once you trigger the progressive downloading of an MP3, there is* ***no way to stop it****. Deleting the* ***Sound*** *object—as you just scripted—will not stop the downloading file. It will just keep downloading regardless of whether you stop the sound, load another sound into the same object, or even delete the entire sound object itself. The only confirmed way to stop the progressive downloading of an MP3 when using Safari is to close the browser window. Unfortunately, that's not really a "realistic" bug workaround, so you'll sadly just have to accept it as a bug and hope that Apple fixes this bug in a future release of Safari. What this means to the viewer who is running Safari is that when he or she clicks the Play button to initiate the progressive downloading and playing of the external MP3 file, if he or she decides halfway through the download to stop the music or to listen to the next track by clicking the Stop or Next Track buttons, respectively, the MP3 track will continue to download. This means that until that track finishes its downloading, most of his or her available bandwidth will be consumed. This will slow down everything else the visitor tries to access/download on your Web site until that downloading file completes. Although this is certainly unfortunate, it's even more of an impetus to try to keep the file sizes of your media down to their—realistically—smallest possible sizes. All other browsers on Mac OS X or Windows are not affected by this issue.*

Next, you need to call the ***stopMusak*** *function from the Stop button so the music stops when it's clicked.*

13. On the **Stage**, double-click the instance of the **mc. toggle play** movie clip symbol.



14. Move the playhead to **Frame 2**. Then, on the **Stage**, single-click the **Stop** button to select it, and open the **Actions** panel (**F9**) if it isn't already open.

```
on (rollOver) {
    _root.helpBubble.text = "stop music";
}

on (rollOut) {
    _root.helpBubble.text = "";
}

on (release) {
    _root.stopMusak();
    prevFrame();
}
```

15. Click at the end of the **`on (release) {`** line, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and type the following:

```
_root.stopMusak();
```

This action executes the ***`stopMusak`*** *function on* ***`_root`*** *(the Scene 1 Timeline). That function deletes the* ***`bgMusak`*** *sound object, which causes the playing sound to stop playing and any MP3 downloading into that* ***`Sound`*** *object to* stop *downloading.*

Last, you need to test the changes you've made to make sure that everything is working as it should.

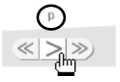
16. Save the changes you've made to **music.fla** by choosing **File > Save** (**Ctrl+S** [Window] or **Cmd+S** [Mac]).

17. Test your movie by choosing **Control > Test Movie** (**Ctrl+Enter** [Windows] or **Cmd+Return** [Mac]).

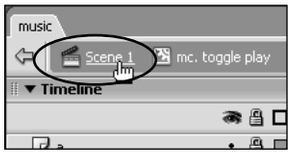
18. When **music.swf** opens in the **Preview** window, you won't hear the music start playing by itself, as it did last time you tested the movie. To start the music playing, click the **Play** button.

19. Once you click the **Play** button, the music will start playing, and the **Play** button will change into a **Stop** button. While the music is playing, if you click the **Stop** button, the music will stop!

So, congratulations! You've just given the viewer the ability to start and stop the background music.

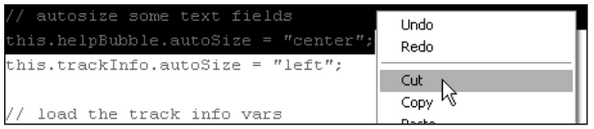


However, while you were testing your movie you probably noticed something strange. When you moved your mouse over the Play and Stop buttons, you probably noticed a small bit of text appear above the buttons. If you remember, the prebuilt ActionScript that was already applied to those buttons told a dynamic text field–on the Stage–with an instance name of ***helpBubble*** *to display the text "play music" and "stop music." Well, the* ***helpBubble*** *dynamic text field hasn't yet been told to automatically resize itself to fit whatever text is displayed inside of it. So that small bit of text you're seeing is only the first letter or two of "play music" and "stop music." In the next step, you're going to use the* ***autoSize*** *action–which you've used previously–to instruct the* ***helpBubble*** *text field to resize itself when necessary.*



20. Close the **Preview** window to return to **music.fla**. Then, return to **Scene 1** by clicking its tab above the top left of the **Timeline**.

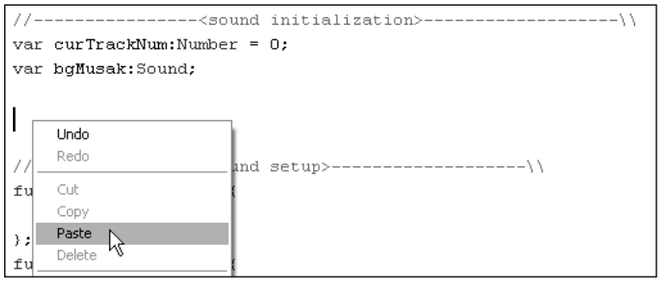
21. Click **Keyframe 1** in layer **a** and open the **Actions** panel (**F9**).



22. In the **Actions** panel, scroll down a little until you reach the commented-out, prebuilt actions for Chapter 10. Select the following actions–toward the top of the prebuilt actions:

```
// autosize some text fields
this.helpBubble.autoSize = "center";
```

23. Then, **right-click** (Windows) or **Ctrl+click** (Mac) anywhere on those selected actions. From the contextual menu that appears, choose **Cut**.



24. Scroll up to the top of the **Actions** palette and click at the end of the **`var bgMusak:Sound;`** action. Press **Enter** (Windows) or **Return** (Mac) twice to create a couple line breaks, and then **right-click** (Windows) or **Ctrl+click** (Mac) that second empty line break you just created. From the contextual menu that appears, choose **Paste**.

This will paste the actions that you just cut from the prebuilt action set.

```
//----------------<sound initialization>-------------------\\
var curTrackNum:Number = 0;
var bgMusak:Sound;

// autosize some text fields
this.helpBubble.autoSize = "center";

//----------------<sound setup>-------------------\\
```

*Now you have an action–just like you've created in earlier chapters–that tells the **`helpBubble`** dynamic text field to automatically resize itself to accommodate any text that gets inserted into it.*

---

25. Save the changes you've made to **music.fla** by choosing **File > Save** (**Ctrl+S** [Window] or **Cmd+S** [Mac]).

26. Test your movie by choosing **Control > Test Movie** (**Ctrl+Enter** [Windows] or **Cmd+Return** [Mac]).

27. When the **music.swf Preview** window opens, move your mouse over the **Play** button. When you do, you'll notice that–above the buttons–the text "play music" appears. Click the **Play** button to play the music and to toggle the **Play** button to show the **Stop** button. When your mouse is over the **Stop** button, the text should then read "stop music."

28. When you're finished checking out the rollover text, close the **music.swf Preview** window.

And voila! You've just completed the construction of the music Play and Stop buttons, as well as some descriptive text that gets inserted into a text field when the viewer rolls over those same buttons.

In the next exercise, you will write a little ActionScript that will pull the ID3 information from the currently playing MP3 and insert it into a text field on the Stage. Just when you thought it couldn't get more exciting… ;-)

5. Displaying the ID3 Information

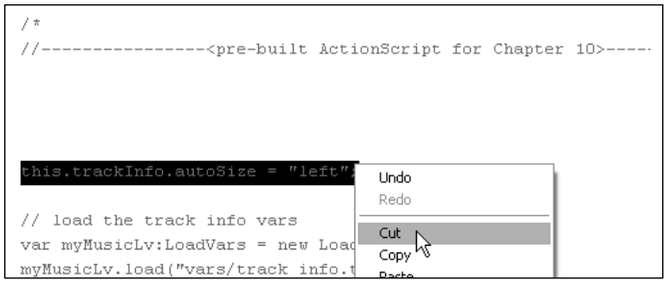
Most of you are probably aware that the "digital music revolution" is well underway. With MP3-playing software such as Apple iTunes for your computer, portable MP3 devices such as the Apple iPod, and now boom boxes and car stereos that play MP3s, it's quickly becoming a technology that is being integrated into our daily lives. If you've ever played an MP3 on your computer, or on a portable MP3-playing device like an Apple iPod, have you ever wondered how it knew that you were listening to "Windowlicker" by Aphex Twin? Or which order the album tracks are supposed to be in, or even which album the songs came from? This information is actually written into each MP3 track, and the various bits of information are called **ID3 tags**. The ID3 tags keep track of a wide variety of information about each MP3 such as the name of the artist, the name of the album, the track name, the year the song was released, which track number it is out of the total number of tracks, the genre, and more.

Starting with Flash MX, you could use a little ActionScript to have the Flash Player grab some of those ID3 version 1 (v1) tags and display them. This is useful for when you want to have an MP3 player, for example, and display some dynamic information about the currently playing track without having to manually type it all out in your FLA. However, the problem with ID3 v1 tags and building an MP3 player that progressively downloaded its MP3 tracks off of a remote server was that ID3 v1 tags are stored at *the end* of the MP3 file. This meant that you couldn't access those ID3 tags until the MP3 had been *completely* downloaded. Obviously, when attempting to create a streaming MP3 player, having the ID3 tags at the end of the file doesn't help you out very much. With the introduction of Flash MX 2004, the Flash Player 7 now supports ID3 v2.3 and 2.4 tags. The great thing about ID3 v2.3 and 2.4 tags is that they're stored at *the beginning* of the MP3 file and can therefore be accessed as soon as the MP3 file starts to download. So if you're building an MP3 player that progressively downloads its tracks–as you are doing in this chapter–you can display those ID3 tags right from the get-go.

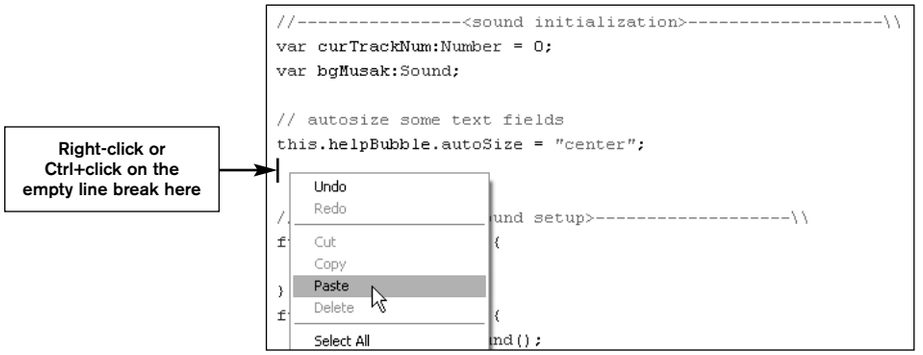
In this exercise, you're going to write some ActionScript that will grab a few of those ID3 tags and display them in the **`trackInfo`** dynamic text field on the Stage.

1. First, make sure that **music.fla** is the current foreground FLA, that you have **Keyframe 1** in layer **a** selected, and that you have the **Actions** panel open (**F9**).

The ID3 tags will be displayed in the long, horizontal dynamic text field at the bottom of the Stage. If you remember, that text field has an instance name of ***`trackInfo`*** *already applied to it. When you use ActionScript to insert some text in that field, it will* not *automatically resize itself to fit the text that gets inserted into it. Just like you did in the previous exercise, you need an action to do that for you.*



2. Scroll down in the **Actions** panel a bit until you come to the action **`this.trackInfo.autoSize = "left";`**. Once you've located that action, select it and then **right-click** (Windows) or **Ctrl+click** (Mac) it. From the contextual menu that appears, choose **Cut**.



3. Scroll back toward the top of the **Actions** panel, click at the end of the **`this.helpBubble.autoSize = "center";`** line, and press **Enter** (Windows) or **Return** (Mac) once to create a line break. Then, **right-click** (Windows) or **Ctrl+click** (Mac) that new, empty line break. From the contextual menu that appears, choose **Paste.**

```
// autosize some text fields
this.helpBubble.autoSize = "center";
this.trackInfo.autoSize = "left";
```

*As you have seen a few times before, you now have an action that sets the **trackInfo** dynamic text field so that it will automatically resize as needed, while keeping the text within it aligned to the left side of the field.*

*If your brain isn't jello yet, you might recall that right before Exercise 4 is a table called "Methods, Properties, and Event Handlers of the Sound Class." In that table is a **Sound** class event handler called **onID3**. The **onID3** event handler is fired every time the ID3 tags change of whichever sound is being loaded using the **bgMusak** sound object. That sounds like just the thing you need to populate a dynamic text field with the MP3's ID3 tags! In fact, that's exactly what you're going to use in these next steps.*

```
//----------------<sound setup>-------------------\\
function stopMusak() {
    delete bgMusak;
};
function playMusak() {
    bgMusak = new Sound();
    bgMusak.onID3 = function() {
    bgMusak.onLoad = function(success) {
        if (!success) {
            trackInfo.text = "Failed to load track.";
        }
    };
    bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);
};
//----------------</sound setup>------------------\\
```

4. In the **Actions** panel, locate the actions contained within the **<sound setup>** comments. Then, click after the **`bgMusak = new Sound();`** line, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and type the opening action for this event handler:

```
bgMusak.onID3 = function() {
```

*Just like the other **Sound** class event handlers—such as **onLoad**, which you wrote in Exercise 3—the **onID3** event handler needs to be assigned to a function. The actions you place within this function will be executed each time the **onID3** event handler is fired.*

```
//----------------<sound setup>-------------------\\
function playMusak() {
    bgMusak = new Sound();
    bgMusak.onID3 = function() {
        trackInfo.text = "artist : " + bgMusak.id3.TCOM + " | track : " + bgMusak.id3.TIT2;
    bgMusak.onLoad = function(success) {
        if (!success) {
            trackInfo.text = "Failed to load track.";
        }
    };
    bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);
};
```

5. Click at the end of the **`bgMusak.onID3 = function() {`** line, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and then type–all on one line with no line breaks–the following:

```
trackInfo.text = "artist : " + bgMusak.id3.TCOM + " | track : " + bgMusak.id3.TIT2;
```

*In plain English, this action reads, "Insert some text into the **`trackInfo`** dynamic text field. The text to insert should be 'artist :' then the ID3 tag TCOM (which is the composer/artist), the text '| track :' and then the ID3 tag TIT2 (which is the track title)." So if the currently loading/playing MP3 is "Windowlicker" by Aphex Twin, the final result of this action–and thereby the text to be inserted into the **`trackInfo`** text field–would be "artist : Aphex Twin | track : Windowlicker".*

***Note:** To see a full list of the ID3 tags that Flash MX 2004 can access, open the **Help** panel (**F1**), and from the Help "books" on the left, open **ActionScript Dictionary > S > Sound.id3**.*

```
//----------------<sound setup>-------------------\\
function stopMusak() {
    delete bgMusak;
};
function playMusak() {
    bgMusak = new Sound();
    bgMusak.onID3 = function() {
        trackInfo.text = "artist : " + bgMusak.id3.TCOM + " | track : " + bgMusak.id3.TIT2;
    };
    bgMusak.onLoad = function(success) {
        if (!success) {
            trackInfo.text = "Failed to load track.";
        }
    };
    bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);
};
//----------------</sound setup>-------------------\\
```

6. Close the **`bgMusak.onID3`** function by clicking at the end of the action that you wrote in the last step, pressing **Enter** (Windows) or **Return** (Mac) to create a line break, and typing the following:

```
};
```

And that's it! All that this action is doing is every time the ID3 information changes for the sound being targeted by the ***bgMusak*** *sound object, the* ***onID3*** *event handler is triggered and some text, along with some ID3 tags, are inserted into the* ***trackInfo*** *text field. As you saw when writing this action, when you want to get access to the ID3 tags for the MP3 that is being downloaded/played, you first type the name of the* ***Sound*** *object that the MP3 is loading into (****bgMusak*** *in this case),* ***id3****, and then the name of the ID3 tag that you want to grab the value of. So if you wanted to get access to any comments that had been inserted into the ID3 comment tag of an MP3, you could simply type* ***bgMusak.id3.COMM****.*

There are also quite a few programs that allow you to edit the ID3 tags of MP3 files. In Appendix B, "Macromedia Flash MX 2004 Resources*," you'll find some recommended ID3 editing programs for both Windows and Mac.*

Before congratulating yourself, you should test your movie to make sure everything is working.

---

7. Save the changes you've made to **music.fla** by choosing **File > Save** (**Ctrl+S** [Window] or **Cmd+S** [Mac]).

---

8. Test your movie by choosing **Control > Test Movie** (**Ctrl+Enter** [Windows] or **Cmd+Return** [Mac]).

9. When **music.swf** appears in the **Preview** window, click the **Play** button. As you have already seen before, this will start playing the music. But when it starts playing, you'll also see some text appear in the **trackInfo** text field. It will say "artist : DJ Shane | track : Taking the Subway." The text "artist :", the pipe (|), and "track :": were all manually specified in the action you wrote in Step 5, but the text "DJ Shane" and "Taking the Subway" is the content of the ID3 tags TCOM and TIT2, respectively. Pretty darn cool!

As has been mentioned previously, any time you can use dynamic content, go for it. By incorporating the ID3 tags into the track information that is displayed in the ***trackInfo*** *dynamic text field, it makes the MP3 player so much easier to update in the future. Whenever you want to add or remove MP3s, either you or the client simply have to verify that the correct ID3 tags have been set, upload the MP3s to the server, and you're done!*

At this point, you might be looking at the large, blank space between the last track, the Play/Stop button, the Next Track button, and where the ID3 information is, and wondering why its there. Later in this chapter, you will be referred to watch the movie ***music_progress.mov*** *where you will learn how to script a MP3 preloader and a playback progress slider. This will sit in that empty space.*

---

10. When you're finished checking out the ID3 text, close the **Preview** window and return to **music.fla**.

In the next exercise, you're going to script a way for the viewer to skip to the next track or go back to the previous track. You will also script the capability for the MP3 player to automatically advance to the next MP3 track in sequence when the currently playing song reaches its end.

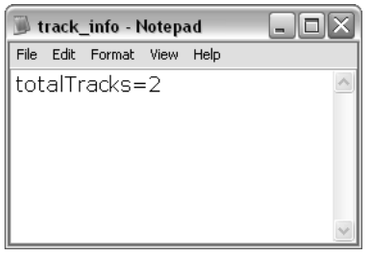
---

6. Changing Tracks

What would an MP3 player jukebox be if you couldn't switch tracks, eh? In this exercise, you will add some ActionScript that will allow the viewer to advance to the next MP3 track in sequence, go to the previous MP3 track, or just wait for the currently playing MP3 to end, and let the MP3 player switch to the next track automatically. Well, what are you waiting for? ;-)

Similar to the slideshow, the MP3 player will loop through the MP3 tracks. When Track 1 is finished playing, the MP3 player will advance to Track 2. When Track 2 has finished playing, the MP3 player will go back to Track 1. (There are only two MP3 tracks total.) So, for the script to know if the MP3 it is currently playing is the *last* MP3 track in the sequence, you need a way to keep track of the total number of MP3 tracks. When constructing the slideshow, you used the variable **`totalFrames`** to keep track of the total number of slides. In this exercise, you're going to create a variable named **`totalTracks`** to keep track of the total number of MP3 tracks. But just like you did with the slideshow, you're going to keep that variable in an external text file, and use a **`LoadVars`** object to load that variable into your SWF. This allows you or the client to easily add or remove MP3 tracks to the MP3 player and make the required changes without even having to open Flash MX 2004. So the first step is to load that **`totalTracks`** variable.

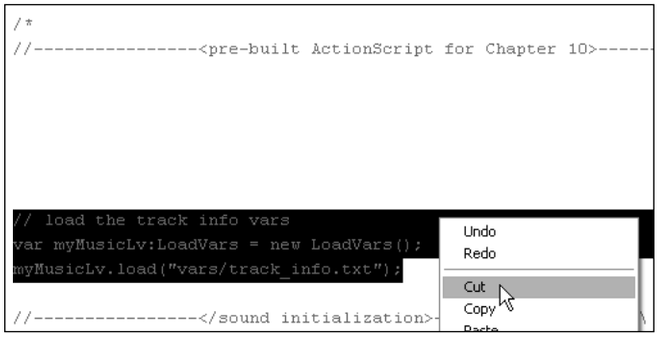
1. Hide or minimize Flash MX 2004, and on your hard drive navigate to your **Desktop > la_eyeworks > site > vars** folder.



2. In the **vars** folder, open the text file **track_info.txt** in a simple text program such as **Notepad** (Windows), **TextEdit** (Mac), or **BBEdit** (Mac). In that text file, you'll see one variable, **`totalTracks`**, with a value of **2**. If you—or the client—were to ever add or remove MP3 tracks, you'd simply have to change that number accordingly.

*Now that you've seen where the text file is, what it is named, and what's inside of it, you can write the **`LoadVars`** script to load it into **music.swf**.*

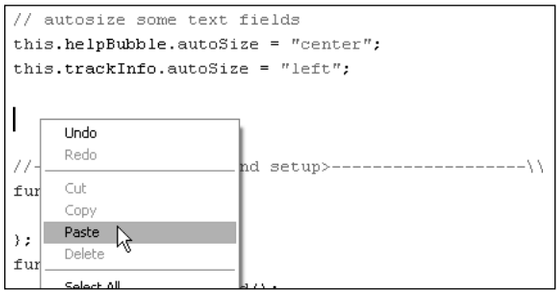
3. Close the text file and return to Flash MX 2004. Then, make sure **music.fla** is the foreground FLA, that you have **Keyframe 1** selected in the **a** layer, and that you have the **Actions** panel open.



4. Scroll down in the **Actions** panel until you reach the following script:

```
// load the track info vars
var myMusicLv:LoadVars = new LoadVars();
myMusicLv.load("vars/track_info.txt");
```

Then, select those three lines, and **right-click** (Windows) or **Ctrl+click** (Mac) anywhere on that selected script. From the contextual menu that appears, choose **Cut**.



5. Scroll to the top of the **Actions** panel, click at the end of the **`this.trackInfo.autoSize = "left";`** line, and press **Enter** (Windows) or **Return** (Mac) twice to create a couple line breaks. Then, **right-click** (Windows) or **Ctrl+click** (Mac) that second empty line break you just created. From the contextual menu that appears, choose **Paste**.

```
// autosize some text fields
this.helpBubble.autoSize = "center";
this.trackInfo.autoSize = "left";

// load the track info vars
var myMusicLv:LoadVars = new LoadVars();
myMusicLv.load("vars/track_info.txt");
```

*Presto! You now have a **LoadVars** object called **myMusicLv** that you're using to load the variables from the file **track_info.txt** located in the **vars** folder. As you saw earlier, this text file contains the variable **totalTracks** with a value of **2**. So whenever you're writing an action, and you need to get access to that variable/value, you can simply type **myMusicLv.totalTracks**.*

Now that you know how many tracks there are total, you can use that number in a script to get the MP3 player to loop through all the music tracks. In the next few steps, you're going to write a script that instructs the MP3 player to advance to the next track in sequence when the currently playing track finishes. If the last track is the currently playing track, when it finishes playing, the script will automatically instruct the MP3 player to load and play the first track.

*Because you will later write a script that changes the track when the current track finishes playing, you need to have a way to detect when a playing track reaches its end. Fortunately, just like **onLoad** and **onID3**, Flash MX 2004 has an event handler to fill this need as well. In the next few steps, you're going to use the **onSoundComplete** event handler to detect when the current track has finished playing and to then advance to the next track in sequence.*

```
//----------------<sound setup>-------------------\\
function stopMusak() {
    delete bgMusak;
};
function playMusak() {
    bgMusak = new Sound();
    bgMusak.onSoundComplete = function() {
    bgMusak.onID3 = function() {
        trackInfo.text = "artist : " + bgMusak.id3.TCOM + " | track : " + bgMusak.id3.TIT2;
    };
    bgMusak.onLoad = function(success) {
        if (!success) {
            trackInfo.text = "Failed to load track.";
        }
    };
    bgMusak.loadSound("mp3s/mp3-" + curTrackNum + ".mp3", true);
};
//----------------</sound setup>------------------\\
```

6. Within the **<sound setup>** comments, click at the end of the **bgMusak = new Sound();** line, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and type the following:

```
bgMusak.onSoundComplete = function() {
```

*Similar to the **onLoad** and **onID3** event handlers that you wrote earlier in this chapter, this line is the first step in creating the **onSoundComplete** event handler. Where the **onLoad** event handler was triggered when the sound was loaded, and the **onID3** event handler was triggered when the sound's ID3 information changed, the **onSoundComplete** event handler is triggered when the currently playing sound finishes (complete). So now you just need to specify what should occur when that happens.*

The first thing you need to find out is if the currently playing track is the last *track in sequence. Because if it is the last track that just finished playing, you need to tell the MP3 player to load and play the* first *track.*

```
//----------------<sound setup>-------------------\\
function stopMusak() {
    delete bgMusak;
};
function playMusak() {
    bgMusak = new Sound();
    bgMusak.onSoundComplete = function() {
        if (curTrackNum == (myMusicLv.totalTracks - 1)) {
```

7. Click at the end of the **`bgMusak.onSoundComplete = function() {`** line, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and type the following:

```
if (curTrackNum == (myMusicLv.totalTracks - 1)) {
```

*Simply, the beginning of this **if** statement reads, "If the track number (the number of the currently playing MP3) is equal to the total number of tracks minus one…"*

*"Why the total number of tracks minus 1?" some of you might be asking. Remember, the MP3s start with the number **0** (for reasons stated in the slideshow chapter) as in **mp3-0.mp3**, **mp3-1.mp3**, and so forth. So even though there are two tracks total, the last MP3 in sequence ends with **1**. Therefore, when you need to determine if the last MP3 track is playing, you need to subtract one from the **totalTracks** variable. Some of you, upon hearing that explanation, might be thinking "Well why not just put **1** in the **totalTracks** variable instead of **2**? Then you'd avoid having to do that silly subtraction stuff." And even though that would surely work, do you really want to explain to the client in charge of updating the **track_info.txt** file to subtract one from the total number of MP3 tracks? Because I don't exactly relish explaining details like that to the client (who has other things to worry about), it's much easier to just subtract one from the **totalTracks** variable in the code.*

*The **if** statement you just wrote essentially asks if the currently playing track is the last track in sequence. If this statement results in the Boolean **true**, when this track finishes playing (**onSoundComplete**), the **curTrackNum** value should be set back to **0**. As you'll see later, after the **curTrackNum** number has been set to whichever track should play next, you'll simply call the **playMusak** function, which will then play whichever MP3 track number is specified in that **curTrackNum** variable.*

```
//----------------<sound setup>-------------------\\
function stopMusak() {
    delete bgMusak;
};
function playMusak() {
    bgMusak = new Sound();
    bgMusak.onSoundComplete = function() {
        if (curTrackNum == (myMusicLv.totalTracks - 1)) {
            curTrackNum = 0;
        } else {
            curTrackNum ++;
        }
```

8. Click at the end of the action you wrote in the last step, press **Enter** (Windows) or **Return** (Mac) once to create a line break, and type the following:

```
curTrackNum = 0;
} else {
curTrackNum ++;
}
```

Note: *Although you will need to create the line breaks just like you see it in the code above, Flash MX 2004 will automatically handle the indentation of the actions for you.*

Taken along with the ***if*** *statement you started writing in Step 7, this script reads, "If the currently playing track number is equal to the total number of available MP3 tracks (meaning, that there aren't any more new tracks to play), set the current track number back to 0 (the beginning). Otherwise, set the current track number to whichever number comes next." So now you've set that* ***curTrackNum*** *to whichever track number should be playing when the current track finishes, but you haven't (yet) instructed the MP3 player to actually* play *that specific MP3 track.*

```
//----------------<sound setup>-------------------\\
function stopMusak() {
    delete bgMusak;
};
function playMusak() {
    bgMusak = new Sound();
    bgMusak.onSoundComplete = function() {
        if (curTrackNum == (myMusicLv.totalTracks - 1)) {
            curTrackNum = 0;
        } else {
            curTrackNum ++;
        }
        playMusak();
    };
```

9. Press **Enter** (Windows) or **Return** (Mac) to create a new line break under the last closed curly brace you typed in Step 8, and type the following:

```
playMusak();
};
```

This instructs the MP3 player to execute the ***playMusak*** *function you built earlier, which, of course, starts playing whichever track number is specified in the* ***curTrackNum*** *variable. The last line* (**};**) *closes the* ***bgMusak.onSoundComplete*** *function, and you now have an MP3 player that will automatically and continually loop through all the MP3 tracks that it has available.*

Now, you just need to cross your fingers, whisper a small prayer, put on your tin foil hat, and test your movie!

---

10. Save the changes you've made to **music.fla** by choosing **File > Save** (**Ctrl+S** [Window] or **Cmd+S** [Mac]).

---

11. Test your movie by choosing **Control > Test Movie** (**Ctrl+Enter** [Windows] or **Cmd+Return** [Mac]).

12. When **music.swf** appears in the **Preview** window, click the **Play** button to start playing the music. Now you just need to kick back and wait for the track to reach the end to hear it automatically advance to the next track in sequence. Coolio!

When the track automatically changes to the next track, you'll see the ID3 text automatically change as well. This is because the ***onID3*** *event handler received the message that the ID3 information for the music had changed, and then executed all the actions within the function it is attached to. The actions within the* ***onID3*** *event handler function grabbed the new ID3 information from the new MP3 and displayed it in the* ***trackInfo*** *dynamic text field. It's like a well-oiled machine!*

Because the script you just wrote essentially instructs the MP3 player to play the next MP3 track in sequence—even jumping back to the first track if it needs to—the MP3 player will play continual music until the cows come home.

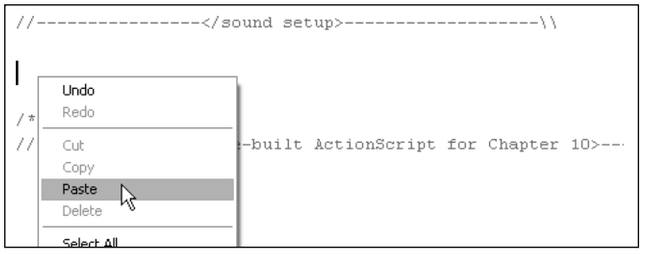
But what if the viewer doesn't want to wait for the currently playing track to end before she can listen to the next track? Maybe the viewer thinks the current track is boring, and wants to hear what the next track sounds like. In that case, the viewer will click the Next Track or Previous Track buttons, that you've kindly given them, to jump one track ahead or one track behind in sequence. In the next steps, you're going to bring all the ActionScript together to make that functionality work. Thankfully, the ActionScript to add that functionality onto the Next Track and Previous Track buttons will be nearly identical to the script you just wrote.

---

13. Close the **Preview** window and return to **music.fla**. Make sure you have **Keyframe 1** selected in layer **a** and that you have the **Actions** panel open.

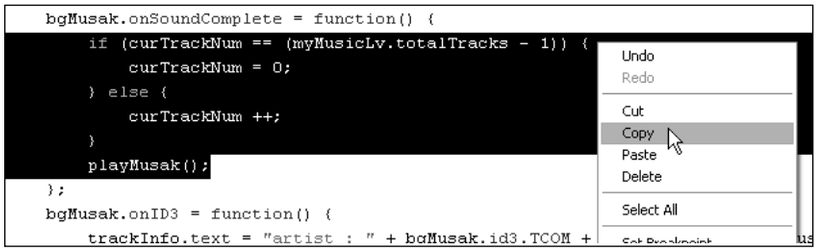
14. Scroll down toward the bottom of the **Actions** panel, and select the giant block of code that starts with the comment `//----------------<next track>----------------\\` and ends with `//----------------</previous track>----------------\\` as you can see in the screen shot above. Then, **right-click** (Windows) or **Ctrl+click** (Mac) anywhere on that selected script. From the contextual menu that appears, choose **Cut**.

Although it may initially look daunting, the ActionScript you just cut is just a series of simple ***onRollOver****,* ***onRollOut****, and* ***onRelease*** *mouse event commands for the Next Track and Previous Track buttons. After you paste this ActionScript into the correct place in the next step, you will add to that script to make it functional for the Next Track and Previous Track buttons.*

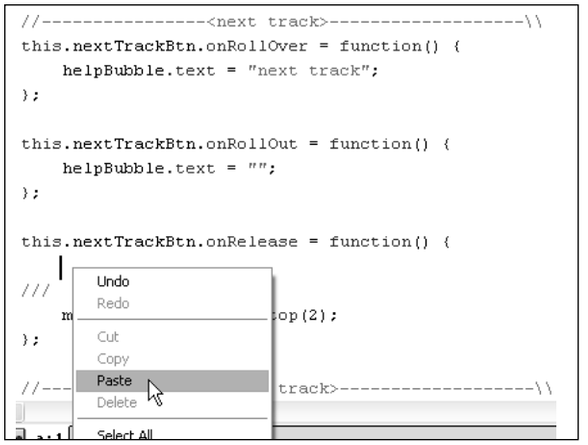


15. Scroll up in the **Actions** panel a little, click at the end of the **`//----------------</sound setup>----------------\\`** line, and press **Enter** (Windows) or **Return** (Mac) twice to create a couple line breaks. Then, **right-click** (Windows) or **Ctrl+click** (Mac) that second empty line break you just created. From the contextual menu that appears, choose **Paste**.

This will paste all that ActionScript you just cut from the prebuilt actions area. Again, this code was provided for you because it's all stuff you've already learned. If you read through the code, you'll see that it's just a series of simple mouse events for the Next Track and Previous Track buttons. What you need to do now, however, is to modify the ***`onRelease`*** *mouse events for both the Next Track and Previous Track buttons so that when the viewer clicks one those buttons, it changes the currently playing track. As was mentioned previously, the ActionScript to create that functionality will be nearly* identical *to the ActionScript you already wrote that changes the track when the currently playing song finishes. So to save yourself some time and energy, you're just going to copy some ActionScript that you've already written and paste it into the* ***`onRelease`*** *states of the Next Track and Previous Track buttons.*



16. Underneath the **`bgMusak.onSoundComplete = function() {`** line, select all the ActionScript that you wrote in Steps 7 through 9, as you can see in the screen shot above. Then **right-click** (Windows) or **Ctrl+click** (Mac) anywhere on that selected script. From the contextual menu that appears, choose **Copy**.



17. Scroll down a little in the **Actions** panel until you come to the **<next track>** comment lines. **Note:** The actions within the **<next track>** and **</next track>** comments are actions that apply to the **Next Track** button on the **Stage**. Then, click at the end of the **`this.nextTrackBtn.onRelease = function() {`** and press **Enter** (Windows) or **Return** (Mac) to create a line break. **Right-click** (Windows) or **Ctrl+click** (Mac) that new line break you just made, and from the contextual menu that appears, choose **Paste**.

```
this.nextTrackBtn.onRelease = function() {
            if (curTrackNum == (myMusicLv.totalTracks - 1)) {
            curTrackNum = 0;
        } else {
            curTrackNum ++;
        }
        playMusak();
///
    musakToggle.gotoAndStop(2);
};
```

*This will paste the **`if`** statement you copied from the **`bgMusak.onSoundComplete`** function. Now, within the **`nextTrackBtn.onRelease`** mouse event, the script essentially reads like this, "When the viewer clicks the Next Track button, if the currently playing track number is equal to the total number of tracks, set the track number back to 0. Otherwise, set the current track number to be one greater than whatever it currently is. After you've set the track number to be whatever it should be, tell the music to play."*

```
this.nextTrackBtn.onRelease = function() {
        if (curTrackNum == (myMusicLv.totalTracks - 1)) {
            curTrackNum = 0;
        } else {
            curTrackNum ++;
        }
```

Note: *Depending on how you copied and pasted that* **`if`** *statement, the first line of that* **`if`** *action might be indented one tabbed space too much. If this happens to you, simply click right before the* **`if`**, *and press* ***Delete*** *or* ***Backspace*** *to remove that extra tab space. When you're finished, the* **`if`** *should be aligned with the* **`}`** *before* **`else`** *and the* **`}`** *under* **`curTrackNum ++;`**.

However, if a viewer comes to the L.A. Eyeworks Web site, clicks the Play button to start playing some music, decides that she doesn't like the music—that's still downloading to her computer—and clicks the Next Track button, the next track will start downloading, but the previous track does not stop downloading. It continues to download, using up unnecessary bandwidth until it finishes its download. This would be horrible because unless the viewer is on a high-speed broadband Internet connection that has bandwidth to spare, his or her poor Internet connection is now trying to down-load two *large MP3 files* simultaneously. *Just like you did with the Stop button, you need to tell the* **`bgMusak`** *music to stop downloading whichever MP3 track it's currently downloading* before *it starts downloading another. Fortunately, you've already created the* **`stopMusak`** *function to do just that.*

```
this.nextTrackBtn.onRelease = function() {
        if (curTrackNum == (myMusicLv.totalTracks - 1)) {
            curTrackNum = 0;
        } else {
            curTrackNum ++;
        }
        stopMusak();
        playMusak();
///
    musakToggle.gotoAndStop(2);
};
```

18. Click at beginning of the **`playMusak();`** line and press **Enter** (Windows) or **Return** (Mac) to create a line break. Then, click the new line break you just created above **`playMusak();`** and type the following:

```
stopMusak();
```

As you used on the Stop button earlier in this chapter, the **`stopMusak`** *function deletes the* **`bgMusak`** *sound object. By deleting the* **`bgMusak`** *sound object, not only does it stop the currently playing sound, but it will also stop any sound that is currently being downloaded into that* **`Sound`** *object as well. Then, after that has occurred, the next action* **`playMusak`** *triggers the next sound to start playing.*

Next, you need to write similar functionality for the Previous Track button. However, where the ***if*** *statement for the Next Track button was* identical *to the* ***if*** *statement you wrote earlier into the* ***bgMusak.onSoundComplete*** *event handler, the* ***if*** *statement for the Previous Track button will need to be changed to accommodate for going* backwards *in sequence.*

Currently, the ***if*** *statement added to the Next Track button first checks to see if the currently playing track is the last available track. However, when going backwards through the MP3 player, you're not really interested if you're on the* last *track. You're more interested if you're on the* first *track. That way, if you* are *on the first track when the viewer clicks the Previous Track button, you can instruct the MP3 player to load the last track.*

```
//----------------<previous track>-------------------\\

this.prevTrackBtn.onRollOver = function() {
    helpBubble.text = "previous track";
};

this.prevTrackBtn.onRollOut = function() {
    helpBubble.text = "";
};

this.prevTrackBtn.onRelease = function() {
    |
///
    musakToggle.gotoAndStop(2);
};

//----------------</previous track>------------------\\
```

19. Scroll down a little in the **Actions** panel until you come to the **<previous track>** comment lines. These actions within the **<previous track>** and **</previous track>** comments are actions that apply to the **Previous Track** button on the **Stage**. Then, click at the end of the **`this.prevTrackBtn.onRelease = function() {`** and press **Enter** (Windows) or **Return** (Mac) to create a line break.

```
this.prevTrackBtn.onRelease = function() {
    if (curTrackNum == 0) {
        curTrackNum = (myMusicLv.totalTracks - 1);
    } else {
        curTrackNum --;
    }
///
    musakToggle.gotoAndStop(2);
};
```

20. Click the new line break you created in the last step and type the following:

```
if (curTrackNum == 0) {
curTrackNum = (myMusicLv.totalTracks - 1);
} else {
curTrackNum --;
}
```

As usual, besides the line breaks you see in the code sample and in the screen shot above, Flash MX 2004 will automatically format the ActionScript for you as you type it. So you don't need to worry about adding or removing tab spaces–Flash MX 2004 will do it all for you.

*Although this **if** statement script is similar to the one you added to the Next Track button, it's a wee bit different. Taken along with the **prevTrackBtn.onRelease** mouse event, this **if** statement reads in plain English, "When the viewer clicks the Previous Track button, if the currently playing track is the first track, set the current track number to be whatever the total number of tracks is. Otherwise, just set the current track number to be one number lower than whatever it currently is." In essence, the **if** statements handle the MP3 player in the same way that the slideshow behaved. With the slideshow, it would continually loop through the 10 available slides as the viewer clicked the Next Slide and Previous Slide buttons. The MP3 player does the same thing, except with MP3s instead of JPGs.*

*To finish, you still need to write the action that deletes the **bgMusak** sound object and another action that starts playing the correct track.*

```
this.prevTrackBtn.onRelease = function() {
    if (curTrackNum == 0) {
        curTrackNum = (myMusicLv.totalTracks - 1);
    } else {
        curTrackNum --;
    }
    stopMusak();
    playMusak();
///
    musakToggle.gotoAndStop(2);
};
```

21. Click after the closed curly brace that you ended with in the last step, and press **Enter** (Windows) or **Return** (Mac) to create a line break. Then, type the following:

```
stopMusak();
playMusak();
```

And–drum roll please–there you have it! Of course, the last step is to save and test your work to make sure everything is working as it should.

---

22. Save the changes you've made to **music.fla** by choosing **File > Save** (**Ctrl+S** [Window] or **Cmd+S** [Mac]).

---

23. Test your movie by choosing **Control > Test Movie** (**Ctrl+Enter** [Windows] or **Cmd+Return** [Mac]).

24. When **music.swf** opens in the **Preview** window, click the **Play** button to start playing the music.

25. When the music starts playing, click the **Next Track** button. You'll notice that the MP3 player will start playing the next track, and the ID3 information will update as well.

26. Once you've verified that the **Next Track** button works correctly, click the **Previous Track** button. You'll notice that the MP3 player will play the previous track in sequence. Although, with only two MP3 tracks, it's hard to tell if it went back a track or went forward a track. I suppose you'll just have to take my word for it. ;-)

If you continually click the Previous Track or Next Track buttons, you'll notice that the MP3 player loops continually through the available tracks.

Hooray! You've constructed a working, fully functional MP3 player complete with Stop, Play, Next Track, and Previous Track buttons, as well as a dynamic text field that is updated with ID3 tags pulled from the MP3 files themselves. Break out the champagne—or at least a spritzer—because that's quite an accomplishment you've achieved in this chapter.

MOVIE | Creating an MP3 Preloader and Displaying the Playback Progress

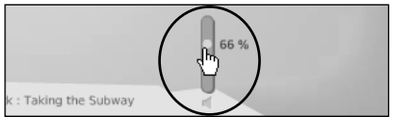
Located on the **H•O•T CD-ROM** in the **movies** folder are four QuickTime movies titled **music_progress.mov**, **music_progress_part2.mov**, **music_progress_part3.mov**, and **music_progress_part4.mov**.

In these movies, you will learn how to build a few different pieces of functionality that fit into one graphic. You will learn how to build a preloader (different from the **`MovieClipLoader`**-based preloader that you constructed in Chapter 8, "*Building a Preloader*") that monitors and displays the downloading progress of the MP3s. Built into that, you will also construct the ActionScript that animates a small slider across that same graphic. This slider represents the playback progress of the currently playing MP3.

You can see the results of these QuickTime movies by pointing your Web browser to **http://www.lynda.com/flashbtb/laeyeworks/**, clicking the **Play** button on the MP3 player, and watching the graphic (highlighted in the screen shot above) to the right of the Play/Stop, Next Track, and Previous Track buttons.

MOVIE | Changing the Volume

Located on the **H•O•T CD-ROM** in the **movies** folder are three QuickTime movies titled **volume_slider.mov**, **volume_slider_part2.mov**, and **volume_slider_part3.mov**.



In these QuickTimes movies, you will learn how to adjust the volume of the music by creating a draggable slider. As you drag the slider up and down, the volume of the music will adjust accordingly.

You can see what you will be building in these movies, by pointing your Web browser to **http://www.lynda.com/flashbtb/laeyeworks/**, clicking the **Play** button on the MP3 player, clicking the small speaker icon toward the right side of the MP3 player bar, and dragging the light-green slider. You'll also notice that as you drag the slider, displayed to right of the slider is the volume listed as a percentage. When you stop dragging the slider, the percentage text disappears. Clicking the **speaker** icon again will hide the volume slider.

*I'd highly recommend watching, and following along with, the **music_progress.mov** and **volume_slider.mov** series of movies. Not only do they teach some new and interesting ActionScript concepts, but you'll also get to add some great functionality to the MP3 player. Treat the movies just as you would exercises in the book.*

*In the next chapter, you will learn how to create a video player. In building the video player, you'll learn how to use the **NetStream** class to progressively download a FLV (**Fl**ash **V**ideo) file off of a remote server, as well as how to stop and play that video. There's lots of new topics in the next chapter, so go rest up and I'll meet you there!*